webtechniques

Object Serialization in Java 1.1

Making Objects Persistent

By Chad Darby

Java 1.1 introduces object serialization, an easy-to-use and easy-to-implement data-storage method that eliminates worries about file formats. Object serialization lets you save an object's state, thereby saving the values of the data members and functionality of the methods. In effect, this provides object "persistence." Say, for example, you are writing a computer game with a save option. If the player's state information -- position, score, and so on -- is maintained in a game object, you can simply serialize that object to a disk file. The serialization will recursively save all objects pointed to by the game object; then, when the player restarts the game, only the game object must be reloaded.

In this article we will step through object serialization using a simple class, then focus on its implementation in a real-world application. (For more on object serialization, see Bruce Eckel's July 1997 "Java Alley" column in *Web Techniques*.)

`Customer` Class Design

To illustrate serializing an object, we will use a simple `Customer` class that holds a customer's name and age; see Figure 1. The class will override the `toString()` method to provide a string representation; however, the real meat of this example will be in the `load()` and `save()` methods, which provide the actual code for reading and writing serialized objects.

To read or write the state of an object to or from a stream, the class must implement the `java.io.Serializable` interface. This interface does not contain any methods or static data members; it is merely a tag indicating that the class can be serialized. Listing One contains the skeleton class declaration of `Customer`.

Writing Objects to a Stream

Writing objects to a stream resembles writing primitives to output streams and involves the following steps:

1. Create a `FileOutputStream` object for the data file.
2. Create an `ObjectOutputStream` using the specified `FileOutputStream`.
3. Write the object to the stream.
4. Close the stream.

The `Customer` class has a `save()` method that will contain the code for writing the object to a stream; see Listing Two.

The `save()` method takes as its parameter a filename that is then used to create the object's `FileOutputStream`, which is in turn used to create an `ObjectOutputStream`. The `ObjectOutputStream` writes the object to th stream with the simpl call `oos .writeObject(this);`.

An `IOExcception` will be thrown if the file cannot be open d for writing or if problems are encountered while accessing the object or its fields. The `save()` method's `throws` keyword is used to list the xceptions the method can throw,

forcing the programmer using the `Customer` class to handle the exception(s).

After the object is written, we flush and close the `ObjectOutputStream`, and the object is saved to a file.

To save an instance of the `Customer` class to disk, we simply call its `save()` method and pass a filename string.

Reading Objects from a Stream

Serialization makes it easy to read objects from an object stream: Calling the `readObject()` method returns the state of a saved object, regardless of the format in which it was saved.

Reading an object from an input stream entails the following steps:

1. Creating a `FileInputStream` for the data file.
2. Creating an `ObjectInputStream` using the specified `FileInputStream`.
3. Reading from the input stream.
4. Closing the stream.

The `load()` method takes a filename as a parameter and returns a `Customer` object; see Listing Three. A `FileInputStream` and `ObjectInputStream` are created in a fashion similar to the `save()` method. The statement `tempCustomer=(Customer) ois .readObject();` reads the object from the stream, after which it must be cast to our class. If a compatible class is not found, the `ClassNotFoundException` is thrown. An `IOException` may also be thrown since we are dealing with file I/O. Again, because of the `throws` keyword in the method declaration, the programmer must handle the exceptions `ClassNotFoundException` and `IOException` when calling the `load()` method.

The `load()` method is declared as static and is therefore available class-wide. You can call static methods without referencing any particular object, so they are commonly used for "utility" methods. In this case, `load()` is a utility for the `Customer` class; because it is static, we can easily call it from our driver using `anyCustomer=Customer.load("foo.bar");`. After the object is read, we can close the `ObjectInputStream`.

Putting It All Together

Having added the functionality needed for object persistence to our `Customer` class, we now need a driver to test it. In the driver, we will create one customer object and save it to a file, then load the object from the file and display the contents.

Listing Four gives the code for the Customer Driver. Two objects of `Customer` are declared: `customerA` and `customerB`. We create a new `Customer` object and pass it the initial values `name="Janine"` and `age=24`. Then we save `customerA` to the disk file foo.bar. By saving the object, we are in fact serializing it.

The object is loaded from foo.bar by calling the `load()` method and passing in the filename. The `load()` method returns a `Customer` object, which is assigned to `customerB`. Because `load()` is static, we don't have to reference a particular `Customer` object, only the class name. In this example, the `load()` method actually deserializes the object from the input stream.

Now that we have successfully loaded an object from the input stream, we can call its methods. The `System.out.println()` statement calls the `Customer` object's `toString()` method, which accesses the object's data members and displays its contents. Additional public methods or data can also be referenced.

This simple example of object serialization used the default methods `writeObject()` and `readObject()`; to control the information saved or restored, th y could have been overridd n. Furth rmore, instead of the `Serializable` interface, we could have implemented the `Externalizable` interface, which saves only the class's identity; the class would then have to provide methods to save and restore the cont nts.

A Web-Based Reporting System

Now I'll move on to a real-world example: implementing object serialization to serve as a light-weight database and create dynamic Web pages based on the data stored in the serialized objects.

My intranet object-serialization example takes place at a fictional company called Mikar Tech. The Information Technology (IT) shop at Mikar Tech would like a Web-based collaborative reporting application. The company is in the process of installing a new sales-support system for their four regional divisions, and the Web application will give them the capability to monitor and update the project status.

Key members of the IT shop agreed that the Web application should provide the following functionality:

1. 1. The ability for regional project managers to provide weekly reports via a Web-based form.
2. 2. The ability to view a one-page summary of all weekly reports with "stop-light" status indicators.
3. 3. The ability to drill down through the summary to get detailed information.

Design Solution

A traditional HTML form on the client side will be used to gather data for the weekly reports. The data will then be submitted to a Java-CGI program residing on the Web server; see Figure 2. This Java-CGI program will perform the following:

1. Store the weekly reports based on the form data as a serialized object.
2. Create a dynamic Web page for weekly reports and save to disk.
3. Create a dynamic Web page for summary reports based on all weekly reports submitted and save the page to disk.
4. Return the weekly report page to the user for viewing.

The main driver of the Java-CGI application is the `StatusUpdate` class. `StatusUpdate` makes the high-level calls to produce the functionality outlined in the design. Listing Five is the code fragment for the `main` routine in the `StatusUpdate` class.

The Weekly Report

Each week, the regional project managers will submit a weekly report via a Web-based form. The weekly report is composed of status ratings on project features: client software, database server, Internet connectivity, and user training.

When the manager rates the status of each feature, they will use red, yellow, or green. For example, if the client software has not been installed, this feature will be assigned red. If work is in progress, the feature will be rated yellow. Finally, if a feature is completed, it is rated green. This assignment of color codes gives the ability to provide a stop-light grid for the summary report.

After the project manager enters the data for the weekly report and submits the report, the Java-CGI program will perform the following functions:

1. Stores the weekly report based on the form data as a serialized object.
2. Creates a dynamic Web page for the weekly report and saves it to disk.

The Weekly Report as a Serialized Object

During the object-oriented analysis and design phase, three classes were identified as the heart and soul of the system; see Figure 3 and 4. Each division's weekly report data needs to be stored. A `Division` object will contain the name of the division (Seattle, WA), a pointer to th division's `ProjectManager` and a list of all the project features. By leveraging object serialization, we can easily save the state of this object.

Since we nly had four company divisions to handle, it would have been overkill to use relational-database access. By using object serialization, we did not waste valuable development time creating a scheme to read and write the data to a disk. Object serialization eliminated the need to save program information to flat text

files or add relational-database connectivity.

To save the state of the `Division` object, you need to provide a method for `save()` and `load()` for the `Division` class. Listing Six is code fragment for the `save()` method. Enhancements were made for saving the object to a data directory and a very simple file-locking mechanism was added. (Code for the `load()` method will be discussed in a later section.)

Once the `save()` method for the `Division` object is called, the complete state data for the object is stored in a file titled <division>.division (for example, Seattle.division).

After the division's weekly report is saved as a serialized object, a dynamic Web page can be created. The `Division` class has a `publishWeeklyReport()` method, which actually calls a number of utility routines to build the different portions of an HTML page: heading, navigation bar, body, and the bottom. The heart of this method is in the call to `buildBody()` method. Listings Seven and Eight provide the code for `publishWeeklyReport()` and `buildBody()`, respectively.

The `Feature` class provides a method to `WebString()`, which gives an HTML-formatted string of its data members (see Listing Nine).

The HTML tags are embedded in the string. This string information is returned to the calling method while the string is used to build the weekly report Web page. This Web page is saved to disk as a flat text file. Since this file has HTML tags, it can later be viewed by a Web browser.

The Project Summary Report

The project summary report is a one-page summary of all weekly reports. The summary gives a graphical display of the weekly reports with "stop-light" status indicators. This report is useful for determining the project status of all divisions at a glance. While viewing the summary report, the user can drill down for specific details in problem areas.

Creating the project-summary report involves loading serialized objects for all divisions and generating HTML tables with data from `Division` objects.

Loading serialized objects. To create the summary report, the serialized object for all divisions must first be loaded. This load process is easy thanks to serialized objects. The `load()` method is very similar to what was presented in the "how-to" section of this article. Enhancements were made for saving the object to a data directory and again, we implemented a simple file-locking mechanism (see Listing Ten).

Loading division objects. To load all of the `Division` objects, you can simply write a `for` loop that calls the `load()` method for each one. The `Division` objects will be stored in a `Vector`, which is a growable array. Listing Eleven is the code fragment for loading the `Division` objects. This method is a member of the `StatusUpdate` class, which serves as the main driver for the application.

HTML Table for Summary Report

Basically, the summary report is a table that lists the project data for each division. Every time a new weekly report is submitted, the status grid must update to display the latest information (see Listing Twelve).

The bulk of the code is in the `buildGrid()` method, which actually traverses the vector of `Division` objects and creates an HTML table with the data (see Listing Thirteen).

The normal HTML tag for a table is `<table border=1 width=100%>`. For the first row of the table, we create a heading by listing the names of each `Division` object. Each `Division` object has a `nameToGridString()` method that returns the name of the `Division` object with HTML table tags (for example, `<td>Houston</td>`).

Once the grid heading is created, the data will be listed for each feature of the

project (Client Software, Database Server, Internet Connectivity and User Training). Each `Feature` object has a `statusToGridString()` method that will return a string representation of the status. The actual string has HTML table code to display a GIF image. So, for example, if we had the following data for a feature:

Division Name: Houston
Feature Name: User Training
Feature Status: red

the HTML string returned would look like Example 1. After all of the features are displayed, a final row is added at the bottom to display the date of the last report update.

When users view the HTML table, they will be able to get the project status of all divisions at a glance. If they want details on a specific problem area, clicking on the status image will allow them to drill down to the details on the weekly report created earlier.

Conclusion

Equipped with the knowledge about implementing serialized objects, you can easily integrate the Java object-serialization technology in your next Intranet application. The full source code is available at www.j-nine.com/pubs/Webtech-sep97.

(Get the source code for this article here.)

Chad (shod) is the principal consultant at J9 Consulting, www.j-nine.com, where he specializes in developing server-side Java applications. He can be reached at darby@j-nine.com.

Sys Admin, SD Expo, SD Magazine, Unixreview, Windows Developer Network, New Architect

www4